HALCON Version 5.2

HALCON/C++
User's Manual

December 19, 2001

This user's manual explains how to use the image analysis tool HALCON in C++, version 5.2

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Edition 1	July 1997
Edition 2	November 1997
Edition 3	March 1998
Edition 4	April 1999
Edition 4a	December 2001

Copyright © 1997-2001 MVTec Software GmbH, Munich, Germany

Information concerning HALCON:

`http://www.mvtec.com`

Contents

1 An Introductory Example **5**

2 The HALCON Parameter Classes **7**
2.1 Iconic Objects (`HObject`) 7
2.1.1 Regions (`HRegion`) 8
2.1.2 Region Arrays (`HRegionArray`) 12
2.1.3 Images (`HImage`) 14
2.1.4 Pixel Values (`HPixVal`) 17
2.1.5 Image Arrays (`HImageArray`) 18
2.1.6 Byte Images (`HByteImage`) 20
2.2 Low-level Objects (`Hobject`) 22
2.3 Numerical Parameters 22
2.3.1 The Class `HCtrlVal` 22
2.3.2 The Class `HTuple` 24
2.3.3 The Simple-Mode 26
2.3.4 The Tuple-Mode 26

3 The Class `HWindow` **29**

4 Structure of the Reference Manual **31**

5 Exception Handling **33**
5.1 The Class `HException` 33
5.2 Return Values of low-level HALCON Operators 33

6 Using HALCON/C++ **35**

7 Typical Image Processing Problems **39**
7.1 Thresholding an Image 39
7.2 Edge Detection 39
7.3 Dynamic Threshold 40
7.4 Texture Transformation 40
7.5 Eliminating small Objects 40
7.6 Selecting oriented Objects 41
7.7 Smoothing of Contours 41

Index **43**

List of Figures **44**

Preface

HALCON/C++ is the integration of the image analysis system HALCON into the host language C++. For a detailed description of the operators of the HALCON system the reader is referred to the HALCON reference manual. The integration of user-defined operators into the HALCON system is described in the C-Interface manual[1].
The reader of this user manual should be familar with basic concepts of image analysis and the programming language C++ .

[1]The HALCON C-Interface is truely different from HALCON/C++. The first is the HALCON internal interface of the HALCON operators which you can use, if you want to integrate your algorithms into HALCON, and the latter is the use of HALCON from the programming language C++.

Chapter 1

An Introductory Example

Let's start with a brief sample program before taking a closer look inside HALCON/C++.

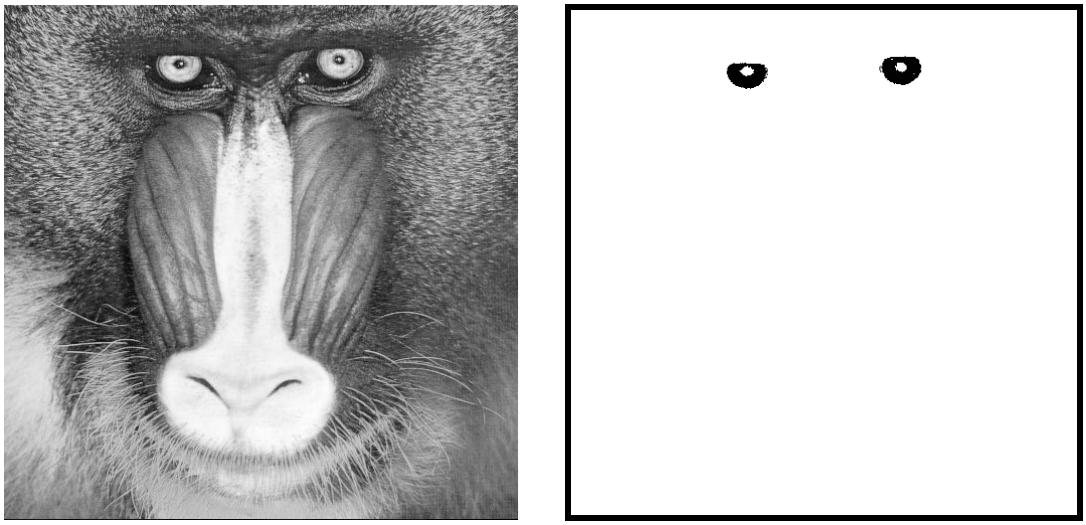

Figure 1.1: The left side shows the input image (a mandrill), and the right side shows the result of the image processing: the eyes of the monkey.

The input image is shown in Fig. 1.1 on the left side. The task is to find the eyes of the monkey by segmentation. The segmentation of the eyes is performed by the C++ program listed in Fig. 1.2, the result of the segmentation process is shown in Fig. 1.1 on the right side.
The program is more or less self-explanatory. However, the basic idea should be explained. First all pixel of the input image are selected where the grayvalues have at least a value of 128, on the assuption that the image Mandrill is a byte image with a grayvalue range between 0 and 255. Second the connected component analysis is performed. The result of the HALCON operator is an array of regions. Each region is isolated in the sense that it does not touch another region according to the neighbourhood relationship. The previous region is divided in a set of regions. Among these regions those two are selected which correspond to the eyes of the monkey. This is done by using shape properties of the regions, the size and the anisometry.
What is the example good for? It seemed to be straight foreward to use the operators listed in the HALCON reference manual in your own C++ program. You don't have to care about the underlying data structures and algorithms, you can ignore specific hardware requirements, if you consider e.g. input and output operators. HALCON handles the memory management

```
#include "HalconCpp.h"

main()
{
  HImage  Mandrill("mandrill");        // read image from file mandrill
  HWindow w;                           // window with size equal to image

  Mandrill.Display(w);                 // display image in window
  w.Click();                           // wait for mouse click

  HRegion Bright = Mandrill >= 128;    // select all bright pixels
  HRegionArray Conn = Bright.Connection(); // get connected components

  // select regions with a size of at least 500 pixels
  HRegionArray Large = Conn.SelectShape("area","and",500,90000);

  // select the eyes out of the instance variable Large by using
  // the anisometry as region feature:
  HRegionArray Eyes = Large.SelectShape("anisometry","and",1,1.7);

  Eyes.Display(w);                     // display result image in window
  w.Click();                           // wait for mouse click
}
```

Figure 1.2: This program extract the eyes of the monkey.

efficiently and hides details from you, and provides an easy to use runtime system. Consider again the program in Fig. 1.2 and ask yourself: isn't image processing fun?
But joking aside. In the following chapters we will discuss the use of the HALCON operators in C++ programs in more detail. Chapter 2 introduces the four different kinds of parameters of the HALCON operators and the corresponding C++ classes. We will explain the passing of tuples of numerical parameters to operators. Using this elegant way of tuples the above `SelectShape()` calls in the example program can be combined into one call in a really natural way. The class `HWindow` is used for the output of images and regions and is explained in Chapter 3. Chapter 4 describes the structure of the reference manual and how to use it. The exception-handling and return values of operators are detailed in Chapter 5. Chapter 6 gives some remarks on the use of the HALCON library from your own application. Templates for solving typical image processing problems are given in Chapter 7. Chapter 7 shows the power of a view lines of HALCON operators.

Chapter 2

The HALCON Parameter Classes

HALCON uses four different kinds of parameters for operators. Not all of them are used in every operator:

- Iconic input objects.
- Iconic output objects.
- Numerical input parameters.
- Numerical output parameters.

Input parameters are passed to an operator *by value*, output parameters are either a return value of an operator or passed to the caller by using the &-operator *by reference*.
Most of the HALCON operators can handle more than one value for a parameter. In the case of iconic objects arrays of the corresponding types (region, image, XLD) are provided and in the case of numerical parameters the tuple concept is used (see `class HTuple` in Section 2.3.2). The operator `Connection()` in the example program in Fig. 1.2 returns an array of iconic objects: `RegionArray`. See the HALCON reference manual if you are unsure whether a parameter of an operator can have more than one value or not. Unfortunaly the standard C++ language doesn't provide polymorphic data structures for container classes like e.g. PROLOG. HALCON/C++ overcomes this by using the class `HTuple` for numerical parameters (see Sections 2.3.2 and 2.3.4). This class implements the correct management of parameters, no matter if the type of the parameter is int, float, string or array.

2.1 Iconic Objects (`HObject`)

Image processing without images is hard to imagine. HALCON provides a data model which means more than handling simple image matrices. The root class of the HALCON/C++ class hierarchy is the virtual class `HObject` which manages entries in the database, i.e., copying, releasing objects. You can use these entries in simple operators by using the class `Hobject`. The classes `HObject` and `Hobject` contain all iconic objects. Important operators like the output (`Display()`) can be applied to all different types in the same manner.
Three classes are derived from the root class `HObject`:

- Class `HImage` for handling images.

- Class `HRegion` for handling regions.
- Class `HXLD` for handling polygons.

These classes are described in detail below. Besides the below listed member-functions the classes contain further functions which are detailed in the HALCON reference manual.

2.1.1 Regions (`HRegion`)

A region is a set of coordinates in the image plane. Such a region need not be connected, and may contain holes. A region can be of a larger size than the actual image format. Regions have a runlength coding in HALCON. The class `HRegion` represents a region in HALCON and contains the following member-functions or *operators* in HALCON terms:

- `HRegion(void)`
  Default constructor. It creates an empty region, i.e., the area of this regions is zero. Not all operators can handle the empty region as input, e.g. some shape property operators.
- `HRegion(const HDChord &line)`
  Constructing a region from a chord. A chord is a horizontal line.
- `HRegion(const HDPoint2D &point)`
  Constructing a region from a discrete 2-dimensional point.
- `HRegion(const HRectangle1 &rect)`
  Constructing a region from a rectangle parallel to the coordinate axis. The coordinates need not be discrete.
- `HRegion(const HRectangle2 &rect)`
  Constructing a region from an arbitrarily oriented rectangle. The coordinates need not be discrete.
- `HRegion(const HCircle &circle)`
  Constructing a region from a circle. The radius and center need not be discrete.
- `HRegion(const HEllipse &ellipse)`
  Constructing a region from an arbitrarily oriented ellipse. The radii and center need not be discrete.
- `HRegion(const char *file)`
  Constructing a region by reading the representation form file. This file can be generated by the member-function `WriteRegion`.
- `HRegion(const HRegion &reg)`
  Copy constructor.
- `HRegion &operator = (const HRegion &reg)`
  Assignment operator.
- `~HRegion(void)`
  Destructor. In contrast to the primitive class (`Hobject`) this class handles the release of memory.

- `void Display(const HWindow &w) const`
  Output of the region in a window.

- `HRegion operator * (double scale) const`
  Zooming the region by an arbitrary factor. The center of scaling is the origin $(0, 0)$.

- `HRegion operator >> (double radius) const`
  `HRegion &operator >>= (double radius)`
  Minkowsky-subtraction of the region with a circle of radius `radius`.

- `HRegion operator << (double radius) const`
  `HRegion &operator <<= (double radius)`
  Minkowsky-addition of the region with a circle of radius `radius`.

- `HRegion operator + (const HDPoint2D &point) const`
  `HRegion &operator += (const HDPoint2D &point)`
  Translating the region by a 2-dimensional point.

- `HRegion &operator ++ (void)`
  Minkowsky-addition of the region with a cross containing five points.

- `HRegion operator + (const HRegion &reg) const`
  `HRegion &operator += (const HRegion &reg)`
  Minkowsky-addition of the region with another region.

- `HRegion operator - (const HRegion &reg) const`
  `HRegion &operator -= (const HRegion &reg)`
  Minkowsky-subtraction of the region with another region.

- `HRegion &operator -- (void)`
  Minkowsky-subtraction of the region with a cross containing five points.

- `HRegion operator ~ (void) const`
  Complement of the region.

- `HRegion operator ! (void) const`
  Transpose the region at the origin.

- `HRegion operator & (const HRegion &reg) const`
  `HRegion &operator &= (const HRegion &reg)`
  Intersection of the region with another region.

- `HRegion operator | (const HRegion &reg) const`
  `HRegion &operator |= (const HRegion &reg)`
  Union of the region with another region.

- `HRegion operator / (const HRegion &reg) const`
  `HRegion &operator /= (const HRegion &reg)`
  Subtract another region from the actual region.

- `HBool operator == (const HRegion &reg) const`
  Boolean test if two regions are identical.

- `HBool operator >= (const HRegion &reg) const`
  `HBool operator > (const HRegion &reg) const`
  `HBool operator <= (const HRegion &reg) const`
  `HBool operator < (const HRegion &reg) const`
  Boolean test if another region is included in the region by using the subset of the corresponding coordinates.

- `double Phi(void) const`
  Orientation of the region by using the angle of the equivalent ellipse.

- `double Ra(void) const`
  Length of the major axis of the equivalent ellipse of the region.

- `double Rb(void) const`
  Length of the minor axis of the equivalent ellipse of the region.

- `long Area(void) const`
  Area of the region, i.e., number of pixels.

- `double X(void) const`
  `double Y(void) const`
  Center point of the region.

- `double Contlength(void) const`
  Length of the contour of the region, see `Contlength()`.

- `double Compactness(void) const`
  Compactness of the actual region, see `Compactness()`.

- `double Anisometry(void) const`
  `double Bulkiness(void) const`
  `double StructureFactor(void) const`
  Shape factors, see HALCON reference manual.

- `double M11(void) const`
  `double M20(void) const`
  `double M02(void) const`
  `double Ia(void) const`
  `double Ib(void) const`
  Moments of the region, see HALCON reference manual.

- `HRectangle1 SmallestRectangle1(void) const`
  Smallest surrounding rectangle parallel to the coordinate axis.

- `HBool In(const HDPoint2D &p) const`
  Boolean test if a point is inside a region.

- `HBool IsEmpty(void) const;`
  Boolean test if the region is empty, i.e., the area of the region is zero.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage     image("mreut");          // Reading an aerial image
  HRegion    region = image >= 190;   // Calculating a threshold
  HWindow    w;                       // Display window
  w.SetColor("red");                  // Set color for regions
  region.Display(w);                  // Display the region
  HRegion    filled = region.FillUp() // Fill holes in region
  filled.Display(w);                  // Display the region
  // Opening: erosion followed by a dilation with a circle mask
  HRegion    open = (filled << 3.5) >> 3.5;
  w.SetColor("green");                // Set color for regions
  open.Display(w);                    // Display the region
  HDPoint2D  trans(-100,-150);        // Vector for translation
  HRegion    moved  = open + trans;   // Translation
  HRegion    zoomed = moved * 2.0;    // Zooming the region
}
```

Figure 2.1: Sample program for the application of the class HRegion.

Figure 2.2: On the left the input image (mreut.tiff), and on the right the region after the opening (open).

A program shows the power of the class HRegion, see Fig. 2.1.
First an aerial image (mreut.tiff) is read from a file. All pixels with a gray value ≥ 190 are selected. This results in one region (region).
This region is transformed by the next steps: all holes in the region are filled (FillUp()) , small parts of the region are eliminated by a morphological operations, first an erosion, a kind of shrinking the region, followed by a dilation, a kind of enlarging the region. The last step is the

zooming of the region. For that the region is first shifted by a translation vector $(-100, -150)$ to the upper left corner and then zoomed by the factor two. Fig. 2.2 shows the input image and the result of the opening operation.

2.1.2 Region Arrays (`HRegionArray`)

The class `HRegionArray` serves as container class for regions. `HRegionArray` has the following member-functions:

- `HRegionArray(void)`
  Constructor for an empty array (`Num()` is 0).
- `HRegionArray(const HRegion &reg)`
  Constructor with a single region.
- `HRegionArray(const HRegionArray &arr)`
  Copy constructor.
- `~HRegionArray(void)`
  Destructor.
- `HRegionArray &operator = (const HRegionArray &arr)`
  Assignment operator.
- `long Num(void)`
  Number of regions in the actual array, largest index is `Num()` $- 1$.
- `HRegion const &operator [] (long index) const`
  Reading the element i of the array. The index is in the range $0 \ldots$ `Num()` $- 1$.
- `HRegion &operator [] (long index)`
  Assigning a region to the element i of the array. The index `index` can be $\geq$ `Num()`.
- `HRegionArray operator () (long min, long max) const`
  Selecting a subset between the lower `min` and upper `max` index.
- `HRegionArray &Append(const HRegion &reg)`
  Appending another region to the region array.
- `HRegionArray &Append(const HRegionArray &reg)`
  Appending another region array to the region array.
- `void Display(const HWindow &w) const`
  Display the regions of the array in a window.
- `HRegionArray operator << (double radius) const`
  Applying the Minkowsky-addition to all regions using a circular mask.
- `HRegionArray operator >> (double radius) const`
  Applying the Minkowsky-subtraction to all regions using a circular mask.

- `HRegionArray operator + (const HRegion &reg) const`
  Applying the Minkowsky-addition to all regions using another region as mask.
- `HRegionArray operator - (const HRegion &reg) const`
  Applying the Minkowsky-subtraction to all regions using another region as mask.
- `HRegionArray operator ~ (void) const`
  Applying the complement operator to each region of the array.
- `HRegionArray operator & (const HRegionArray &reg) const`
  Intersection of each region of the actual array with the union of `reg`.
- `HRegionArray operator | (const HRegionArray &reg) const`
  Union of each region in the actual array with the union of `reg`.
- `HRegionArray operator / (const HRegionArray &reg) const`
  Difference of each region in the actual array with the union of `reg`.

Most HALCON operators accept `HRegionArray` as data structure for the input parameter, e.g. `union()`, `intersection()`, `difference()`, etc. The constructor instantiating the region array `HRegionArray` by a single region `HRegion` makes it possible to handle only one region. Without changing the data structure a `HRegionArray` can be used as input parameter even in the case of a single region.
Fig. 2.3 shows a short example how to use the class `HRegionArray`.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage        image("control_unit");       // Reading an image from file
  // Segmentation by regiongrowing
  HRegionArray  regs = image.Regiongrowing(1,1,4,100);
  HWindow       w;                           // Display window
  w.SetColored(12);                          // Set colors for regions
  regs.Display(w);                           // Display the regions
  HRegionArray  rect;                        // New array
  for (long i = 0; i < regs.Num(); i++)      // For all regions in array
  { // Test size and shape of each region
    if ((regs[i].Area() > 1000) && (regs[i].Compactness() < 1.5))
      rect.Append(regs[i]);                  // If test true, append region
  }
  image.Display(w);                          // Display the image
  rect.Display(w);                           // Display resulting regions
}
```

Figure 2.3: Sample program for use of the class `HRegionArray`.

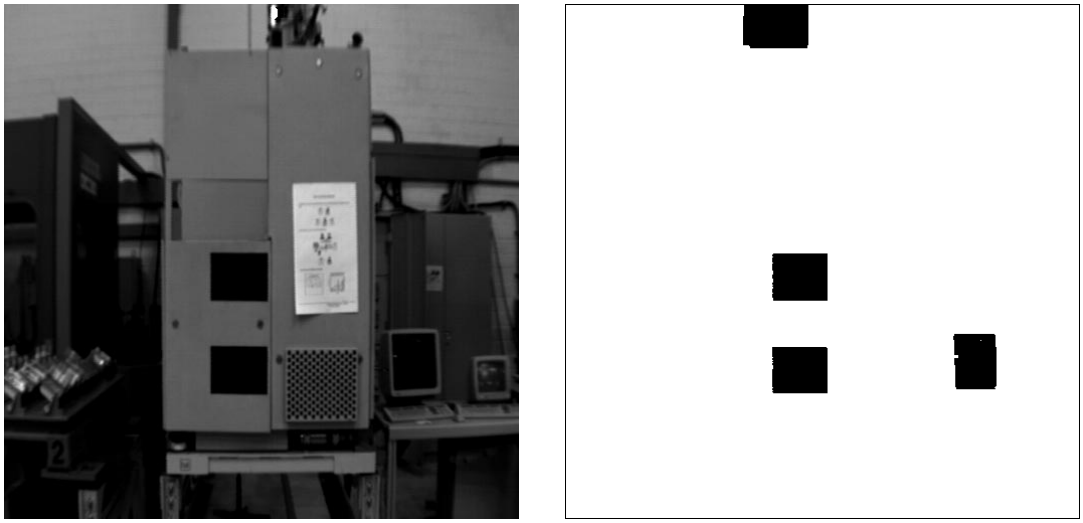

Figure 2.4: On the left side the input image (`control_unit.tiff`), and on the right side the selected rectangles.

The first step is to read an image. In this case it shows a control unit in a manufacturing environment, see Fig. 2.4 on the left side. By applying a regiongrowing algorithm from the HALCON library the image is segmented into regions. Each region inside the resulting region array `regs` is now selected according to its size and its compactness. Each region of a size larger than 1000 pixel and of a compactness value smaller than 1.5 is appended to the region array `rect`. After the processing of the `for` loop only the regions showing on the right side of Fig. 2.4 are left.

2.1.3 Images (`HImage`)

Images contain at least one image matrix in conjuction with a region. This region defines the domain of the image. Various pixel types are supported. The class `HImage` is the root class for all derived image classes. By using the class `HImage` all different pixel types can be handled in a unique way (polymorphism). The class `HImage` is not virtual, thus it can be instantiated. It contains the following member-functions:

- `HImage(void)`
  Default constructor, empty image.
- `HImage(const char *file)`
  Constructing an image by reading from a file, s. `ReadImage()`.
- `HImage(int width, int height, const char *type)`
  Constructing an image of a defined size and a specific pixel type, s. `GenImageConst()`.
- `HImage(void *ptr, int width, int height, const char *type)`
  Constructing an image of a defined size and a specific pixel type by copying memory, s. `GenImage1`.
- `HImage(const HImage &image)`
  Copy constructor.

- `virtual ~HImage(void)`
  Destructor.
- `HImage &operator = (const HImage &arr)`
  Assignment operator.
- `virtual const char *PixType(void) const`
  Return the pixel type of the image.
- `int Width(void) const`
  Return the width of the image.
- `int Height(void) const`
  Return the height of the image.
- `HPixVal GetPixVal(int x, int y) const`
  Access a pixel value via the (x, y) coordinates.
- `HPixVal GetPixVal(long k) const`
  Linear access of a pixel value.
- `virtual void SetPixVal(int x, int y, const HPixVal &val)`
  Set the pixel value via the (x, y) coordinates.
- `virtual void SetPixVal(long k, const HPixVal &val)`
  Set the pixel value by linear access.
- `virtual void Display(const HWindow &w) const`
  Display an image in a window.
- `HImage  operator & (const HRegion &reg) const`
  Reduce the domain of an image, s. `ReduceDomain()`.
- `HImage  operator + (const HImage &add) const`
  Adding two images, s. `AddImage()`.
- `HImage  operator - (const HImage &sub) const`
  Subtracting two images, s. `SubImage()`.
- `HImage  operator * (const HImage &mult) const`
  Multiplication of two images, s. `MultImage()`.
- `HImage  operator - (void) const`
  Inverting the values of the image s. `Invert()`.
- `HImage  operator + (double add) const`
  `HImage  operator - (double sub) const`
  `HImage  operator * (double mult) const`
  `HImage  operator / (double div) const`
  Arithmetic operators, s. `ScaleImage()`.

- `HRegion operator >= (const HImage &image) const`
  Selecting all pixel with gray values brighter than or equal to the input image, s. `DynThreshold()`.

- `HRegion operator <= (const HImage &image) const`
  Selecting all pixel with gray values darker than or equal to the input image, s. `DynThreshold()`.

- `HRegion operator >= (double thresh) const`
  Selecting all pixel with gray values brighter than or equal to a threshold, s. `Threshold()`.

- `HRegion operator <= (double thresh) const`
  Selecting all pixel with gray values darker than or equal to a threshold, s. `Threshold()`.

- `HRegion operator == (double thresh) const`
  Selecting all pixel with gray values equal to a threshold, s. `Threshold()`.

- `HRegion operator != (double thresh) const`
  Selecting all pixel with gray values not equal to a threshold, s. `Threshold()`.

Fig. 2.5 gives an example of the use of the class `HImage`.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage   image("mreut");                    // Aerial image
  HWindow  w;                                 // Output window
  image.Display(w);                           // Display image
  // Returning the size of the image
  cout << "width  = " << image.Width();
  cout << "height = " << image.Height() << endl;
  // Interactive drawing of a region by using the mouse
  HRegion  mask = w.DrawRegion();
  // Reduce the domain of the image to the mask
  HImage   reduced = image & mask;
  w.ClearWindow();                            // Clear the window
  reduced.Display(w);                         // Display the reduced image
  // Applying the mean filter in the reduced image
  HImage   mean = reduced.MeanImage(61,61);
  mean.Display(w);
  HRegion  reg = bild >= (mean + 3);
  reg.Display(w);
}
```

Figure 2.5: Sample program for the use of the class `HImage`.

Figure 2.6: On the left side the input image (`mreut.tiff`), and on the right side the segmented regions in the selected image domain.

The example starts by reading a byte image from a file. The aim is to extract bright parts from the image. The used filter and the segmentation process itself has to be applied only in a pre-chosen part of the image in order to accelerate the runtime. This part is selected by drawing an arbitrary region with the mouse. This region `mask` serves as input for reducing the domain of the original image (`&`-operator). The mean filter with a mask size of 61×61 is applied to the resulting region `reduced`. Bright pixels are selected by applying the `>=` operator. All pixels brighter than the mean filtered part of the image `reduced` $+3$ are selected. Fig. 2.6 shows the result of the sample program in Fig. 2.5.

2.1.4 Pixel Values (HPixVal)

The class `HPixVal` is used for accessing the pixel values of the class `HImage`. Gray values can be set and returned independent of their types:

- `HPixVal(void)`
  Default constructor.
- `HPixVal(const HComplex &Val)`
  Constructing a pixel value from a complex number.
- `HPixVal(int Val)`
  Constructing a pixel value from an integer (`int`).
- `HPixVal(long Val)`
  Constructing a pixel value from a long (`long`).
- `HPixVal(HByte Val)`
  Constructing a pixel value from a byte (`byte`).
- `HPixVal(double Val)`
  Constructing a pixel value from a double (`double`).

- `HPixVal(const HPixVal &Val)`
  Copy constructor.
- `HPixVal &operator = (const HPixVal &grey)`
  Assignment operator.
- `operator HByte(void) const`
  Converting a pixel value to byte $(0 \ldots 255)$.
- `operator int(void) const`
  Converting a pixel value to `int`.
- `operator long(void) const`
  Converting a pixel value to `long`.
- `operator double(void) const`
  Converting a pixel value to `double`.
- `operator HComplex(void) const`
  Converting a pixel value to `Complex`.

The handling of the class `HPixVal` is explained by an example in Fig. 2.7.
The program in Fig. 2.7 inverts the input image. The input image is a byte image. First a copy is generated and the image size is determined. In the first run the pixels are accessed linearly. In the second run the pixel are accessed via the (x, y)-ccordinates.

2.1.5 Image Arrays (`HImageArray`)

The same way which was used to define arrays of regions is used to obtain arrays of images. The class is named `HImageArray` and contains the following member-functions:

- `HImageArray(void)`
  Default constructor: empty array, no element.
- `HImageArray(const HImage &reg)`
  Constructing an image array from a single image.
- `HImageArray(const HImageArray &arr)`
  Copy constructor.
- `~HImageArray(void)`
  Destructor.
- `HImageArray &operator = (const HImageArray &arr)`
  Assignment operator.
- `long Num(void) const`
  Returning the number of elements in the array.
- `HImage const &operator [] (long index) const`
  Reading the element i of the array. The index is in the range $0 \ldots$ `Num()` $- 1$.

```
#include "HalconCpp.h"
#include <iostream.h>

main ()
{
  HImage  in("mreut");             // Aerial image
  HWindow w;                       // Output window
  in.Display(w);                   // Displaying the image
  HImage out = in;                 // Copying the image
  int  width  = out.Width();       // Width of the image
  int  height = out.Height();      // Height of the image
  long end    = width * height;    // Number of pixel of the image

  // 1. run: linear accessing
  for (long k = 0; k < end; k++) {
    int pix = in.GetPixVal(k);     // Reading the pixel
    out.SetPixVal(k,255-pix);      // Setting the pixel
  }
  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();

  // 2. run: accessing the image via the coordiantes (x,y)
  for (int y=0; y<height; y++) {
    for (int x=0; x<width; x++) {
      int pix = in.GetPixVal(x,y); // Reading the pixel
      out.SetPixVal(x,y,255-pix);  // Setting the pixel
    }
  }
  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();
}
```

Figure 2.7: Sample program for the use of the class HPixVal.

- `HImage &operator [] (long index)`
  Assigning a region to the element i of the array. The index `index` can be $\geq$ `Num()`.
- `HImageArray operator () (long min, long max)`
  Selecting a subset between the lower `min` and upper `max` index.
- `HImageArray &Append(const HImage &image)`
  Appending another image to the image array.
- `HImageArray &Append(const HImageArray &images)`
  Appending another image array to the image array.

2.1.6 Byte Images (`HByteImage`)

An important specialization of the class `HImage` is the class `HByteImage`. The range of the pixel values of the class `HByteImage` is between 0 and 255. This pixel type covers more than 90% of all applications in the field of image processing. The advantage of the class `HByteImage` in comparison to the class `HImage` is the simplified access to the pixel values. This is because the class `HPixVal` is not neccesary. Besides the member-functions of `HImage` the class `HByteImage` contains the following extensions:

- `HByteImage(void)`
  Default constructor.
- `HByteImage(const char *file)`
  Constructing a byte image by reading a file.
- `HByteImage(int width, int height)`
  Constructing an empty byte image of a given size.
- `HByteImage(HByte *ptr, int width, int height)`
  Constructing a byte image by copying memory.
- `HByteImage(const HByteImage &image)`
  Copy constructor.
- `virtual ~HByteImage(void)`
  Destructor.
- `HByte &operator[] (long k)`
  Setting a pixel value by linear accessing.
- `HByte operator[] (long k) const`
  Reading a pixel value by linear accessing.
- `HByte &operator() (long k)`
  Setting a pixel value by linear accessing.
- `HByte operator() (long k) const`
  Reading a pixel value by linear accessing.
- `HByte &operator()(int x, int y)`
  Setting a pixel value by accessing it via (x, y) coordinates.
- `HByte operator()(int x, int y) const`
  Reading a pixel value by accessing it via (x, y) coordinates.
- `HByteImage operator & (int i)`
  Applying the logical “and”-operation on each pixel with `i`.
- `HByteImage operator << (int i)`
  Applying a left-shift on each pixel with `i`.
- `HByteImage operator >> (int i)`
  Applying a right-shift on each pixel with `i`.

- `HByteImage operator ~ (void)`
  Complement of each pixel.
- `HByteImage operator & (HByteImage &ima)`
  Pixel by pixel logical “and”-operation of two images.
- `HByteImage operator | (HByteImage &ima)`
  Pixel by pixel logical “or”-operation of two images.
- `HByteImage operator ^ (HByteImage &ima)`
  Pixel by pixel logical “xor”-operation of two images.

The advantage of the class `HByteImage` can be seen when accessing each pixel, see Fig. 2.8.

```
#include "HalconCpp.h"
#include <iostream.h>

main ()
{
  HByteImage  in("mreut");         // Aerial image
  HWindow w;                       // Output window
  in.Display(w);                   // Displaying the image
  HImage out = in;                 // Copying the image
  int  width  = out.Width();       // Width of the image
  int  height = out.Height();      // Height of the image
  long end    = width * height;    // Number of pixel of the image

  // 1. run: linear accessing
  for (long k = 0; k < end; k++)
    out[k] = 255 - in[k];          // Reading and setting the pixel

  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();

  // // 2. run: accessing the image via the coordinates (x,y)
  for (int y=0; y<height; y++)
    for (int x=0; x<width; x++)
      out(x,y) = 255 - out(x,y);   // Reading and setting the pixel

  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();
}
```

Figure 2.8: Sample program for accessing a pixel value using the class `HByteImage`.

The class `HPixVal` is not neccesary in this example. Furthermore the member-functions `GetPixVal` and `SetPixVal` are not used. `HByteImage` allows the accessing of pixel values

in a notation like in the programming language C. The result of the example in Fig. 2.8 is basically the same as in the example in Fig. 2.7. The program in Fig. 2.8 is shorter, easy to read, and has a better runtime performence.

2.2 Low-level Objects (`Hobject`)

For dealing with the low-level objects HALCON/C++ provides the data type class `Hobject`. This data type allows you to access the internal HALCON data management. The class handles the keys of the database. The class `Hobject` serves as basis for the class `HObject` and the derived classes. The class `Hobject` has the following member-functions:

- `Hobject(void)`
  Default constructor.
- `Hobject(const Hobject &obj)`
  Copy constructor.
- `virtual ~Hobject(void)`
  Destructor.
- `Hobject &operator = (const Hobject &obj)`
  Assignment operator.
- `void Clear(void)`
  Freeing the memory, but preserving the key.

2.3 Numerical Parameters

HALCON/C++ can handle different types of numerical parameters for HALCON operators:

- discrete numbers (`long`),
- floating point numbers (`double`), and
- strings (`char *`).

If a HALCON operator returns numerical parameters and it is not clear how many parameters are returned, then the parameter has to be of the type class `HTuple`. In this case *all* other numerical output parameters have to be of the type class `HTuple`.
The type of numerical input parameters is simply the class `HTuple`. This is because `HTuple` provides constructors for all basic data types including `HTuple`.

2.3.1 The Class `HCtrlVal`

Before considering the different ways of passing values to numerical parameters the classes `HTuple` and `HCtrlVal` are discribed. The class `HCtrlVal` serves as basis for the class `HTuple` and is normally hidden from the user because it is only used temporarily for type conversion:

- `HCtrlVal(void)`
  Default constructor.
- `HCtrlVal(long l)`
  Constructing a value from `long`.
- `HCtrlVal(int l)`
  Constructing a value from `int`.
- `HCtrlVal(double d)`
  Constructing a value from `double`.
- `HCtrlVal(const char *s)`
  Constructing a value from `char *`.
- `HCtrlVal(const HCtrlVal &v)`
  Copy constructor.
- `~HCtrlVal(void)`
  Destructor.
- `HCtrlVal& operator = (const HCtrlVal &v)`
  Assignment operator.
- `int ValType() const`
  Type of a value.
- `operator int(void) const`
  Conversion to `int`.
- `operator long(void) const`
  Conversion to `long`.
- `operator double(void) const`
  Conversion to `double`.
- `operator const char*(void) const`
  Conversion to `char *`.
- `double D() const`
  Accessing a value and conversion to `double`.
- `long L() const`
  Accessing a value and conversion to `long`.
- `int I() const`
  Accessing a value and conversion to `int`.
- `const char *S() const`
  Accessing a value and conversion to `char *`.
- `HCtrlVal operator + (const HCtrlVal &val) const`
  Adding two values.

- `HCtrlVal operator - (const HCtrlVal &val) const`
  Subtracting two values.

- `HCtrlVal operator * (const HCtrlVal &val) const`
  Multiplying two values.

- `HCtrlVal operator / (const HCtrlVal &val) const`
  Division of two values.

2.3.2 The Class `HTuple`

The class `HTuple` is built by using the class `HCtrlVal`. The class `HTuple` implements an array of dynamic length for instances of the class `HCtrlVal`. The default constructor constructs an empty array (`Num() == 0`). This array can dynamically be expanded via assignments. The memory management, i.e., reallocation, freeing, is also managed by the class. The index for accessing the array is in the range between 0 and `Num()` $- 1$.
The class `HTuple` plays an importent role for the export of programs written in HDevelop to C++ code. The following member-functions reflect only a small portion of the total. For further information please see the file `HTuple.h` which is included in the HALCON distribution.

- `HTuple(void)`
  Default constructor. Constructs an empty tuple.

- `HTuple(long l)`
  Constructing an array of length 1 from a discrete number `long` at index position 0.

- `HTuple(int l)`
  Constructing an array of length 1 from a discrete number converted to the internal type `long` at index position 0.

- `HTuple(HCoord c)`
  Constructing an array of length 1 from a coordinate at index position 0.

- `HTuple(double d)`
  Constructing an array of length 1 from a floating number `double` at index position 0.

- `HTuple(const char *s)`
  Constructing an array of length 1 from a string `char*` at index position 0.

- `HTuple(const HTuple &t)`
  Copying a tuple.

- `~HTuple()`
  Destructor.

- `HTuple &operator = (const HTuple& in)`
  Assignment operator.

- `HTuple Sum(void) const`
  Adding all elements in case they are numbers.

- `HCtrlVal &operator [] (int i)`
  Setting the $i-$th element.
- `HCtrlVal operator [] (int i) const`
  Reading the $i-$th element.
- `HTuple operator + (const HTuple &val) const`
  Adding two tuples element by element. The arrays have to be of the same size.
- `HTuple operator + (double &val) const`
  `HTuple operator + (int &val) const`
  Adding a number from each element of the tuple.
- `HTuple operator - (const HTuple &val) const`
  Subtracting two tuples element by element. The arrays have to be of the same size.
- `HTuple operator - (double &val) const`
  `HTuple operator - (int &val) const`
  Subtracting a number to each element of the tuple.
- `HTuple operator * (const HTuple &val) const`
  Multiplying two tuples element by element. The arrays have to be of the same size.
- `HTuple operator * (double &val) const`
  `HTuple operator * (int &val) const`
  Multiplying a number with each element of the tuple.
- `HTuple operator / (const HTuple &val) const`
  Division of two tuples element by element. The arrays have to be of the same size.
- `HTuple operator / (double &val) const`
  `HTuple operator / (int &val) const`
  Division of each element of the tuple by a number.
- `HTuple Concat(const HTuple &t) const`
  Concatenating two tuples.
- `extern ostream& operator<<(ostream &s, const HTuple &t)`
  Output of a tuple.
- `extern istream& operator>>(istream &s, HTuple &t)`
  Input of a tuple.

Fig. 2.9 shows a short sample how to use tuples, i.e., the class `HTuple`.
The default constructor generates an empty tuple. By assigning values to the tuple it is automatically expanded, and the data types of the values are also stored. For accessing the tuple the normal array notation can be used. If the data type of a value is not known in advance, an explicit type conversion has to be performed, s. Fig. 2.9.

```
#include "HalconCpp.h"
#include <iostream.h>

main ()
{
  HTuple  t;
  cout << t.Num() << '\n';            // The length of the tuple is 0
  t[0] = 0.815;                       // Assigning values to the tuple
  t[1] = 42;
  t[2] = "HAL";
  cout << t.Num() << '\n';            // The length of the tuple is 3
  cout << "HTuple = " << t << '\n';   // Using the << operator
  double d = t[0];                    // Accessing the tuple, if the
  long   l = t[1];                    // the types of the elements
  char  *s = t[2];                    // are known
  // Accessing the tuple, if the types of the elements are not known
  printf("Values: %g %ld %s\n",t[0].D(),t[1].L(),t[2].S());
}
```

Figure 2.9: Sample for the use of the class `HTuple`.

2.3.3 The Simple-Mode

By applying the simple-mode the HALCON operators can be used in your own C++ programs in a very natural way. All numerical output values are variables having the following data types:

- `long` for discrete numbers,
- `double` for floating point numbers, and
- `char*` for strings.

The values are passed *by reference* using the &-operator. The data type string is a pointer to `char`. *The user has to take care of the memory allocation for the data type string in the case of output numerical values.*
Examples for calling a HALCON operator in the simple-mode can be found in Chapter 7.

2.3.4 The Tuple-Mode

The tuple concept has been mentioned several times in this manual. A lot of HALCON operators have numerical parameters, and each parameter may have more than just one value. The class `HTuple` has been introduced in order to handle such parameters in a highly efficient way. Separate calls to one HALCON operator can be combined to just one call by using the tuple concept.
Besides the above simple-mode HALCON/C++ offers also the tuple-mode. If a numerical output parameter uses a tuple of values, the tuple-mode has to be applied. A mixture between simple- and tuple-mode is not possible. Furthermore the tuple-mode has to be used if the *type* or the *number* of the returned values of a HALCON operator is not known in advance.

The syntax of tuple- and simple-mode are basically the same. The data type in the tuple-mode is just `HTuple`, that's all.
If you are not interested in a certain value of a numerical output parameter you can use the anonymous variable "_" instead of passing a "dummy" tuple to that variable.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HTuple SysFlags,Info;                          // Tuple variables
  long  i;                                       // Loop variable

  ::get_system("?",&SysFlags);                   // Get system values

  for (i=0; i<SysFlags.Num(); i++) {
     ::get_system(SysFlags[i].S(),&Info);        // Get i-th Sysflag
     out << SysFlags[i] << "=" << Info << "\n";  // Print i-th Sysflag
   }
}
```

Figure 2.10: A sample program for using the tuple-mode: output of the actual HALCON system state.

The sample program in Fig. 2.10 shows the use of a tuple. The program obtains information on the actual HALCON system state. The call `::get_system("?",&SysFlags)` gets all flags of the system and its current values. Because the number and the type of the numerical output parameter are not known in this case, the call has to be made in the tuple-mode. The rest of the program is self-explanatory.

Chapter 3

The Class `HWindow`

Another importent class for building programs with HALCON/C++ is the class `HWindow`. This class provides the management of HALCON windows in a very convenient way. The properties of HALCON windows can be easily changed, images, regions, and polygons can be displayed, etc. The class contains the following member-functions:

- ```
  HWindow(int Row=0, int Column=0,
          int Width=-1, int Height=-1,
          int Father = 0, const char *Mode = "",
          const char *Host = "")
  ```
  Default constructor. The constructed window is opened.

- `~HWindow(void)`
  Destructor. This closes the window.

- `void Click(void) const`
  Waiting for a mouse click in the window.

- `HDPoint2D GetMbutton(int *button) const`
  Waiting for a mouse click in the window. It returns the current mouse position in the window and the number of the button that was pressed.

- `HDPoint2D GetMbutton(void) const`
  Waiting for a mouse click in the window. It returns the current mouse position in the window.

- `HDPoint2D GetMposition(int *button) const`
  Returning the mouse position and the pressed button without waiting for a mouse click.

- `HDPoint2D GetMposition(void) const`
  Returning the mouse position without waiting for a mouse click.

- `HCircle DrawCircle(void) const`
  Drawing a circle.

- `HEllipse DrawEllipse(void) const`
  Drawing an ellipse.
  ```

- `HRectangle1 DrawRectangle1(void) const`
  Drawing a rectangle parallel to the coordinate axis.

- `HRectangle2 DrawRectangle2(void) const`
  Drawing a rectangle with an arbitrary orientation and size.

Besides those elementary member-functions the class `HWindow` contains more operators which are explained in detail in the HALCON reference manual in the `Graphics` chapter.
Fig. 3.1 shows the typical use of some member-functions of the class `HWindow` and the different possibilities of displaying images and regions.

```
#include "HalconCpp.h"

main ()
{
  HImage  image("control_unit");     // Reading an image from a file
  HWindow w;                         // Opening an appropiate window
  image.Display(w);                  // Display the image
  w.SetLut("change2");               // Set a lookup table
  w.Click();                         // Waiting for a mouse click
  w.SetLut("default");               // Set the default lookup table
  w.SetPart(100,100,200,200);        // Set a part of the window
  image.Display(w);
  w.Click();
  // Adapting the part to the image again
  w.SetPart(0,0,bild.Height()-1,bild.Width()-1);
  image.Display(w);
  HRegionArray regs = image.Regiongrowing(1,1,4,100);
  w.SetDraw("margin");
  w.SetColored(6);
  regs.Display(w);
  w.Click();
  image.Display(w);
  w.SetShape("rectangle1");
  regs.Display(w);
}
```

Figure 3.1: Sample program for the use of the class `HWindow`.

The window is opened after reading the image from a file. This means, the window is scaled to the size of the image. The lookup table is changed afterwards, and the program waits for a mouse click in the window. A part of the image is zoomed now, and the program waits again for a mouse click in the window. By applying a region growing algorithm from the HALCON library (`Regiongrowing()`) regions are generated and displayed in the window. Only the margin of the regions is displayed. It is displayed in 6 different colors in the window. The example ends with another way of displaying the shape of regions. The smallest rectangle parallel to the coordinate axes surrounding each region is displayed.

Chapter 4

Structure of the Reference Manual

The HALCON/C++ reference manual contains the complete description of all operators in the HALCON system. This chapter explains how to read the description of the operators.
Most of the HALCON operators are used with more than one signature. The reasons for this are:

- Operators can be accessed either via the low-level data structure `Hobject` or via the high level classes `HImage`, `HRegion`, `HImageArray`, and `HRegionArray`. The user can choose amongst them.
- Many of operators can have more than one value for a parameter, see Chapter 2. This holds for both iconic objects (images and regions) and for numerical parameters.

The user can choose between the different ways of calling an operator according to his needs. The description in the reference manual reflects the low-level operators. This affects only the way of calling the operator, not the description of the functionality itself. The latter remains the same in any case.
If an operator is called using a low-level class, the return value of that operator is a state value of type `Herror`. The operator itself doesn't belong to a class hierarchy. Examples for such operators are:

```
Herror ::fetch_polygon (Hobject Region,
                        const HTuple& Tolerance,
                        HTuple* Rows, HTuple* Columns)
Herror ::area_center   (Hobject Regions, HTuple* Area,
                        HTuple* Row, HTuple* Column)
Herror ::area_center   (Hobject Regions, long* Area,
                        double* Row, double* Column)
Herror ::class_ndim2   (Hobject Image, Hobject* Regions,
                        const HTuple& NumberOfChannels)
Herror ::threshold     (Hobject Image, Hobject* Region,
                        const HTuple& MinGrey,
                        const HTuple& MaxGrey)
Herror ::mean_image    (Hobject Image, Hobject* MeanImage,
                        const HTuple& MaskWidth,
                        const HTuple& MaskHeight);
```

The operator ::area_center marks the difference between one and several regions. If exactly one region is passed, then size, row, and column refer only to single value. If more than one region is passed, the numerical output parameter has to be of type HTuple because for *each* region a value for size, row, and column is passed.
The use of the member-functions of the classes HRegion, HImage seems to be more complicated if you view it from the reference manual. The reason for this is that values are returned by the member-functions and the operation itself is applied on the instance (this). This causes fewer parameters of the operator. However, the rules for the transformation are very simple:

1. The type of the *first* input parameter determines the class to which the operator belongs as a member-function. This parameter vanishes from the set of parameters of this operator.

2. The first output parameter determines the return value of the operator. Again, this parameter vanishes from the set of parameters.

This is a very convenient way for calling operators from HALCON/C++. Consider this example:

```
HRegion      HImage::Threshold
             (const HTuple &MinGrey, const HTuple &MaxGrey) const

HRegionArray HImageArray::Threshold
             (const HTuple &MinGrey, const HTuple &MaxGrey) const

HImageArray  HImageArray::MeanImage
             (const HTuple &MaskWidth,
              const HTuple &MaskHeight) const
```

Calling the member-functions looks like:

```
HImage  Image("control_unit");
HImage  Mean    = Image.MeanImage(11,11);
HRegion Region  = Mean.Threshold(0,120);
```

Compare the above approach with the use of the low-level data type HError:

```
Hobject Image, Mean, Region;
::read_image(&Image, "control_unit");
::mean_image(Image, &Mean, 11, 11);
::threshold(Mean, &Region, 0, 120);
::clear_obj(Image);
::clear_obj(Mean);
::clear_obj(Region);
```

The memory management has to be done manually by the user.

Chapter 5

Exception Handling

Two techniques are used in HALCON/C++ to handle runtime errors. The first creates an instance of the error and passes it to the exception handler. The second technique can be used when dealing with low-level operators: the return value corresponds to an error which can be analyzed subsequently.

5.1 The Class `HException`

If a runtime error occurs in HALCON/C++ an instance of the class `HException` is created. This instance contains all information concerning the error. After the generation the instance is passed to the exception handler. The exception handler can handle the error. The default exception handler returns an error message and terminates the program. Of course, you can implement and use your own exception handler. The type of the exception handler is:

```
typedef void (*Handler)(const HException &except);
```

The parameter of the handler is the instance of the actual error. The declaration of the class `HException` is shown in Fig. 5.1.
The important members of an error are:

`line`: Number of the program line in which the error occurred.

`file`: Name of the file in which the error occurred.

`proc`: Name of the actual HALCON operator.

`err`: Number of the error, see below.

`message`: Error text.

5.2 Return Values of low-level HALCON Operators

The return values of type `Herror` of the HALCON operators can be divided into two categories:

- messages `H_MSG_*` and
- errors `H_ERR_*`.

```
class HException {
public:
  HException(const char *f, long l, const char *p, Herror e, const char *m);
  HException(const char *f, long l, const char *p, const char *m);
  HException(const char *f, long l, const char *p, Herror e);
  static Handler InstallHHandler(Handler proc);

  static      Handler handler;      // Exception-Handler
  long        line;                 // Line number
  const char  *file;                // File name
  const char  *proc;                // Name of the operator
  Herror      err;                  // Number of the error
  const char  *message;             // Error text
  void        PrintException(void); // Default exception handler
};
```

Figure 5.1: Part of the declaration of the class `HException`.

According to the parameter concept of the HALCON system four different messages can be returned:

- `H_MSG_TRUE`: the operator terminated without an error and the result value is the boolean value true.
- `H_MSG_FALSE`: the operator terminated without an error and the result value is the boolean value false.
- `H_MSG_VOID`: the operator terminated without an error and the result value void is returned.
- `H_MSG_FAIL`: the operator terminated without an error and the result value means the operator has not performed successfully. This means, e.g. an operator is not responsible or a specific situation has not occurred.

In practice nearly all HALCON operators return the message `H_MSG_TRUE` if no error occurs. An error in a HALCON operator normally leads to an exception handling. You can deactivate the exception handler by calling the operator

```
::set_check("~give_error");
```

In this case you can use the operator `::error_text()`. This operator returns for a given error number the corresponding error message. The operator

```
::set_check("give_error");
```

activates the HALCON internal exception handling again.

Chapter 6

Using HALCON/C++

The HALCON distribution contains some examples for building an application with HALCON/C++. The examples can be found in the `HALCONROOT` directory `examples/cpp`. The files `*.mdp` can serve as a template for your own application. A HALCON/C++ application has to be run from a DOS shell because no general framework for a specific interactive application is neccasary. The reason is that you should be able to integrate your own interactive environment and set up your own arrangement.
Currently this project file can be compiled with the Microsoft Visual C++ compiler 5.0. **Please make sure to reserve a stack size of 6 MByte when linking an application with the HALCON libraries.** More information on that topic can be found in the manual Getting Started and in the user's manual of HDevelop.
The HALCON/C++ system consists of the following files:

`include/cpp/HalconCpp.h`: include-file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C interface.

`lib/architecture/libhalconcpp.a/.so/.sl`: The HALCON/C++ library (UNIX).

`lib/architecture/libhalcon.a/.so/.sl`: The HALCON library (UNIX).

`halconcpp.lib/.dll`: The HALCON/C++ library (Windows NT).

`halcon.lib/.dll`: The HALCON library (Windows NT).

`include/cpp/HProto.h`: external function declarations.

`examples/cpp/example1.cpp, ..., examples/C/example11.cpp`: example programs.

`examples/cpp/makefile.*`: Sample makefiles to compile the example programs for several arcitectures.

`images/`: Several images used by the example programs.

`help/english.*`: Files necessary for online informations.

`doc/*`: Various manuals (in subdirectories).

`doc/ps/manual/*`: postscript manuals.

`doc/pdf/manual/*`: postscript manuals.

`doc/html/*`: HTML manuals

`doc/html/`: HALCONoperator reference (HTML).

Your own C++ programs that use HALCON operators, have to include the file `HalconCpp.h`. Do this by adding the include-command:

```
#include "HalconCpp.h"
```

near the top of your C++ file. Using this syntax, the compiler looks for `HalconCpp.h` in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the

```
-I<pathname>
```

command line flag to denote the include-file directory. `HalconCpp.h` contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface.
The library `libhalconcpp.a/.so/.sl` contains the various components of the HALCON/C++ interface. `libhalcon.a/.so/.sl` is the actual HALCON-library. The corresponding NT libraries are `halconcpp.lib/.dll` and `halcon.lib/.dll`.
Finally, there are several example programs in the HALCON/C++ distribution:

`example1.cpp` reads an image and demonstrates several graphics operators.

`example2.cpp` demonstrates the direct pixel access.

`example3.cpp` An example for the usage of pixel iterators.

`example4.cpp` demonstrates the edge detection with a sobel filter.

`example5.cpp` solves a more complicatet problem.

`example6.cpp` is a very simple test programm.

`example7.cpp` demonstrates the generic pixel access.

`example8.cpp` An example for the usage of tuple mode.

`example9.cpp` introduces the XLD structure.

`example10.cpp` demonstrates the usage of several contour structures.

`example11.cpp` is another simple example for the usage uf tuples.

`makefile` sample makefile for `example1.c`.

In the following some important environment variables shall be introduced. In principle, they are the same both under UNIX and Windows NT but in the latter case will be set automatically during the setup. So we will only consider their handling in an UNIX environment.
While a HALCON program is running, it accesses several files (e.g. `help/*`). To tell HALCON where to look for these files, the environment variable `HALCONROOT` has to be set. `HALCONROOT` points to the HALCON home directory. If HALCON is installed in the directory `/usr/halcon`, `HALCONROOT` has to be set accordingly, using the command `setenv HALCONROOT /usr/halcon/` (`csh` syntax). `HALCONROOT` is also used in the sample makefile.
If user-defined packages are used, the environment variable `HALCONEXTENSIONS` has to be set. HALCON will look for possible extensions and their corresponding help files in the directories given in `HALCONEXTENSIONS`. See the HALCON C-Interface Programmer's Manual for more details.
Two things are important in connection with the example programs: The default directory for the HALCON operator `read_image(...)` to look for images is `/usr/halcon/images`. If the images reside in different directories, the appropriate path must be set in `read_image(...)` or the default image directory must be changed, using `set_system("image_dir","/...")`. This is also possible with the environment variable `HALCONIMAGES`. It has to be set before starting the program. An appropriate definition could look like: `setenv HALCONIMAGES /usr/halcon/images`. It is also possible to put the images in several directories. In that case, all the directories (separated by colons under Unix and semicolons under Windows NT) have to be set in `HALCONIMAGES`.

```
PROG_NAME     = example1
DESTINATION   = $(PROG_NAME)
HALCONROOT    =
ARCHITECTURE  =

BIN           = .
SOURCE        = .
H_INCLUDE     = $(HALCONROOT)/include
CPP_INCLUDE   = $(HALCONROOT)/include/cpp
H_LIB         = $(HALCONROOT)/lib/$(ARCHITECTURE)

LIBRARIES     = -lhalconcpp -lhalcon -lm -lX11
CC            = CC -g

$(BIN)/$(DESTINATION) : $(PROG_NAME).o
      $(CC) -o $(BIN)/$(DESTINATION) -L$(H_LIB) \
      $(PROG_NAME).o \
      $(LIBRARIES)

$(PROG_NAME).o: $(PROG_NAME).cpp
      $(CC) -c -I$(H_INCLUDE) -I$(CPP_INCLUDE) $(SOURCE)/$(PROG_NAME).cpp
```

Figure 6.1: sample makefile for `example1.cpp` (compiles program `example1`)

The second remark concerns the output terminal. In the example programs, no host name is

passed to `open_window(...)`. Therefore, the window is opened on the machine that is specified in the environment variable `DISPLAY`. If output on a different terminal is desired, this can be done either directly in `open_window(...,"hostname",...)` or by specifying a host name in `DISPLAY`, like `setenv DISPLAY hostname:0.0` (`csh` syntax).
To end this chapter, Figure 6.1 shows a sample makefile for the program `example1.cpp` (see file `makefile`, too). If necessary, the paths in the makefile have to be replaced by the corresponding paths in your actual system before compiling the example program.
Things are a bit different with VisualC++. Mainly there is no such thing like a makefile. Instead you have to create a new project which has to contain the `halcon.lib/.dll` and the `halconcpp.lib/.dll` files additionally to the desired source file. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the VisualC++ compiler implicitly calls "Update all dependencies" if a new file is added to a project. Since HALCON runs under Unix as well as under Windows NT, the include file `HalconCpp.h` includes several Unix-specific headers as well if included under Unix. Since they don't exist under NT, and the VisualC++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.
Also make sure that the stacksize is sufficient. Some sophisticated image processing problems require up to 6 MB stacksize, so make sure to set the settings of your compiler accordingly (See your compiler manual for additional information on this topic).

Chapter 7

Typical Image Processing Problems

This chapter shows the power of the HALCON system to offer or find solutions for image processing problems. Some typical problems are introduced and sample solutions provided.

7.1 Thresholding an Image

Some of the most common sequences of HALCON operators may look like the following one:

```
HByteImage   Image("file_xyz");
HRegion      Threshold        = Image.Threshold(0,120);
HRegionArray ConnectedRegions = Threshold.Connection();
HRegionArray ResultingRegions =
               ConnectedRegions.SelectShape("area","and",10,100000);
```

This terse program performs the following:

- All pixel are selected with gray values between the range 0 and 120.
- A connected component analysis is performed.
- Only regions of at least 10 pixel size are selected. This step can be considered as a step to remove some of the noise from the image.

7.2 Edge Detection

For the detection of edges the following sequence of HALCON/C++ operators can be applied:

```
HByteImage Image("file_xyz");
HByteImage Sobel = Image.SobelAmp("sum_abs",3);
HRegion    Max   = Sobel.Threshold(30,255);
HRegion    Edges = Max.Skeleton();
```

A brief explanation:

- Before applying the sobel operator it might be useful to apply first a low-pass filter to the image in order to suppress noise.

- Besides the sobel-operator you can also use filters like `EdgesImage`, `Prewitt`, `Robinson`, `Kirsch`, `Roberts`, `BandpassImage`, or `Laplace`.
- The threshold (in our case 30) has to be appropiately selected depending on data.
- The resulting regions are thinned by a `Skeleton()` operator. This leads to regions with a pixel width of 1.

7.3 Dynamic Threshold

Another way to detect edges is e.g. the following sequence:

```
HByteImage Image("file_xyz");
HByteImage Mean      = Image.MeanImage(11,11);
HRegion    Threshold = Image.DynThreshold(Mean,5,"light");
```

Again some remarks:

- The size of the filter mask (in our case 11×11) is corraleted with the size of the objects which have to be found in the image. Indeed the sizes are proportional.
- The dynamic threshold selects the pixel with a positive gray value difference of more than 5 (brighter) than the local environment (mask 11×11).

7.4 Texture Transformation

Texture transformation is useful in order to obtain specific frequency bands in an image. Thus a texture filter detects specific structures in an image. In the following case this structure depends on the chosen filter; 16 are available for the operator `LawsByte()`.

```
HByteImage Image("file_xyz");
HByteImage TT   = Image.LawsByte(Image,"ee",2,5);
HByteImage Mean = TT.MeanImage(71,71);
HRegion    Reg  = Mean.Threshold(30,255);
```

- The mean filter `MeanImage()` is applied with a large mask size in order to smooth the “frequency” image.
- You can also apply several texture transformations and combine the results by using the operators `AddImage()` and `MultImage()`.

7.5 Eliminating small Objects

The following morphological operator eliminates small objects and smoothes the contours of regions.

```
            ...
segmentation(Image,&Seg);
HCircle Circle(100,100,3.5);
HRegionArray Res = Seg.Opening(Circle);
```

- The term `segmentation()` is an arbitrary segmentation operator and results in an array of regions (`Seg`).
- The size of the mask (in this case the radius is 3.5) determines the size of the resulting objects.
- You can choose an arbitrary mask shape.

7.6 Selecting oriented Objects

Another application of morphological operators is the selection of objects having a certain orientation:

```
          ...
segmentation(Image,&Seg);
HRectangle2 Rect(100,100,0.5,21,2);
HRegionArray Res = Seg.Opening(Rect);
```

- Again `segmentation()` leads to an array of regions (`Seg`).
- The width and height of the rectangle determines the minimum size of the resulting regions.
- The orientation of the rectangle determines the orientation of the regions.
- Lines with the same orientation as `Rect` are kept.

7.7 Smoothing of Contours

The last example in this user's manual deals again with morphological operators. Often the margins of contours have to be smoothed for further processing, e.g. fitting lines to a contour. Or small holes inside a region have to be filled:

```
          ...
segmentation(Image,&Seg);
HCircle Circle(100,100,3.5);
HRegionArray Res = Seg.Closing(Circle);
```

- Again `segmentation()` leads to an array of regions (`Seg`).
- For smoothing the contour a circle mask is recommended.
- The size of the mask determines how much the contour is smoothed.

Index

Num(), 12
HTuple, 7
Hobject, 7
AddImage(), 15
Anisometry(), 10
Append(), 12
Append, 19
Area(), 10
Bulkiness(), 10
Click(), 29
Compactness(), 10
Connection(), 7
Contlength(), 10
D(), 23
Display(), 9, 12
DrawCircle(), 29
DrawEllipse(), 29
DrawRectangle1(), 30
DrawRectangle2(), 30
DynThreshold(), 16
FillUp(), 11
GetMbutton(), 29
HByte(), 18
HByteImage, 20
HCtrlVal, 22
HImageArray, 18
HImage, 7, 14, 17
HObject, 7
HPixVal, 17
HRegionArray, 12
HRegion, 8
HTuple, 22, 24
HWindow, 29
HXLD, 8
Hobject, 22
I(), 23
Ia(), 10
Ib(), 10
In(), 10
Invert(), 15
IsEmpty(), 10
L(), 23
M02(), 10
M11(), 10
M20(), 10
MultImage(), 15
Num(), 12, 18
Phi(), 10
Ra(), 10
Rb(), 10
ReduceDomain(), 15
S(), 23
ScaleImage(), 15
SelectShape(), 6
StructureFactor(), 10
SubImage(), 15
Sum(), 24
Threshold(), 16
X(), 10
Y(), 10

Access, 12, 18

Simple-Mode, 26

Tuple-Mode, 26

List of Figures

1.1 The eyes of a monkey 5
1.2 This program extract the eyes of the monkey. 6

2.1 Sample program for the application of the class `HRegion`. 11
2.2 Region processing using a sample aerial image 11
2.3 Sample program for use of the class `HRegionArray`. 13
2.4 Region array processing 14
2.5 Sample program for the use of the class `HImage`. 16
2.6 Image processing using an aerial image. 17
2.7 Sample program for the use of the class `HPixVal`. 19
2.8 Sample program for accessing a pixel value using the class `HByteImage`. 21
2.9 Sample for the use of the class `HTuple`. 26
2.10 A sample program for using the tuple-mode: output of the actual HALCON system state. 27

3.1 Sample program for the use of the class `HWindow`. 30

5.1 Part of the declaration of the class `HException`. 34

6.1 sample makefile for `example1.cpp` (compiles program `example1`) 37